Evolution of Complexity COMP 6026
Assignment 1:"methinks it is like a weasel"
aka "what difference does crossover make?"[i]*

Supervised labs available – check course outline.
Check the assignments website occasionally to see if any updated versions of this assignment have been posted.

This assignment asks you to program a simple evolutionary algorithm from scratch in a language of your choice to investigate the hypothesis that crossover increases the speed of evolution.

*This assignment is not assessed coursework – but you'll find subsequent assignments (which *are* assessed coursework) much easier if you're already familiar with the basics of evolutionary algorithms, as provided by this assignment. You do not need to hand in anything for this assignment.

Contents

- Overview
- Step by step instruction
- Questions

Overview

The scientific question I want you to investigate is: What effect does crossover have on the rate of evolution? To investigate this we will test the hypothesis that a genetic algorithm with crossover finds solutions to a simple optimisation problem faster when using crossover than without crossover. There are many details that need to be filled-in in order to test this hypothesis:

What kind of genetic algorithm?
The easiest kind of evolutionary algorithm to code is a steady state genetic algorithm with tournament selection. You have seen the outline of a basic GA, the steady state variety of GA, and a description of tournament selection in lecture 3.

What kind of crossover?
Uniform crossover will be a sensible crossover operator to test. This is also defined in lecture 3.

What kind of simple optimisation problem?
Here we'll use the "methinks it is like a weasel" model discussed in lecture 1. The task is to evolve a character string that matches this line from Hamlet. You can find an example of a java applet that implements this here: http://home.pacbell.net/s-max/scott/weasel.html together with some explanation. You don't need to use java and you don't need to create a graphical user interface. But you may find that page useful to illustrate what the problem is[ii]. Your fitness function, the measure of how good a proposed solution is, can simply be equal to the number of characters in the candidate solution that match the characters in the target string.

Other parameters and conditions?
Let's use a population size of 500 individuals. Mutation rate of 1/L characters per reproduction on average, where L is the number of characters in the target string.

Tournament size of 2. The speed we'll measure will be the number of fitness evaluations used up to the first occurrence of a perfect match to the 'methinks it is like a weasel' target string. When crossover is used we'll apply it to all reproductions.

That's basically a specification of a set-up that can be used to address the hypothesis we want to test. I don't mind if you vary from these details. So if that all makes sense – off you go, find out whether that hypothesis is correct or not. If that doesn't make sense and you don't know where to start, don't panic. I'll talk you through what needs to be done to below.

Step by step

1) Representing and initialising individuals

The first thing to do is make a representation of an individual. An individual needs to be a string of characters, 28 characters long (the number of characters in the target phrase). Initially each individual in your population will be a random string of characters. So you need to write a routine that creates random strings of 28 characters. You might like to limit the characters to the 26 letters of the alphabet plus spaces, but its probably more straight-forward to allow all ascii characters from 32 to 127 (http://www.lookuptables.com/). It's probably a better idea to represent individuals as an array of characters rather than a real string.

10 examples of random individuals:

```
0.   #0B((NRK6nxUT$h!7<sRwPl1tV
1. 'm}+tj)8!C``g [,Zi1OP>bBsCQH
2. S_0nqi${?'9'`lLVPWfrV{?E0*qm
3. zu-a,  2WC"LjJ;qQe9jL2Je{gLr
4. ,p!y_k-YviBytJ/HM"Yk1Qu=yDxv
5. @Rb|hg(q0!G<CG-D<!6#5QNsi%+T
6. @KWW&mFW2t03*/~`m]?fnd^BYGmi
7. :$YP[1:J]fGZZq1=" z~3YxUSIJ=
8. 34$3VvG^AWE#;(JwwYq!ap,%!Er!
9. wt^^'( Rk^Ske88-ML)Oq n{!x%#
```

2) Evaluating individuals

Now write a routine that evaluates how good an individual is – the fitness function. This needs to go through the string character by character, and at the same time, go through the target string "methinks it is like a weasel" character by character, and simply count how many of the characters in the two strings agree.

e.g. examples of some (hand selected) random individuals with their fitnesses

```
0.   #0B((NRK6nxUT$h!7<sRwPl1tV, fit= 1
1. 'm}+tj)8!C``g [,Zi1OP>bBsCQH, fit= 0
2. S_0nqi${?'9'`lLVPWfrV{?E0*qm, fit= 0
3. zu-a,  2WC"LjJ;qQe9jL2Je{gLr, fit= 1
4. ,p!y_k-YviBytJ/HM"Yk1Qu=yDxv, fit= 1
5. @Rb|hg(q0!G<CG-D<!6#5QNsi%+T, fit= 0
6. @KWW&mFW2t03*/~`m]?fnd^BYGmi, fit= 0
7. :$YP[1:J]fGZZq1=" z~3YxUSIJ=, fit= 0
8. 34$3VvG^AWE#;(JwwYq!ap,%!Er!, fit= 1
17. .!5AhHaR?-Cfee0TarU@R{wga?Bc, fit= 2
38. ~QrhO4L" 'UPNW,t&we!tbPnt4Se, fit= 3
41. d?|+.h`uXJPH&s2_[)[ Cpt06"Vp, fit= 2
```

3) Mutation

Write a routine that takes an individual and returns a ‘mutant’ version of it. That is, a copy that’s the same as the original but for each character, with probability 1/28, change that character to a new random character.

Informal pseudocode:
for i from 1 to L
    if (rand()<(1/L)) then
        <new random character>→ child[i]
    else
        parent[i] → child[i]

e.g.
```
@Rb|hg(q0!G<CG-D<!6#5QNsi%+T
Mutated to
@Rb|hg(q0!G<Ck-D-!6#5QNsi%+T
```

4) A mutation hill-climber

You now have all the parts you need to make a simple mutation hill climber – you don’t need to make a hill-climber to address the hypothesis, but it’s a useful intermediate step to test your components. Here’s some outline pseudocode:

Initialise individual → A.
while (<maximum fitness not found>)
    mutate(A)→B.
    if (fitness(B)>fitness(A)) then B→A.

This will test the basics you’ve prepared so far.

4) A GA without crossover

Now you’re ready to make a simple steady state GA with tournament selection. The main change from the hill climber is that you need to make a population of individuals. You also need to be able to pick an individual at random from this population.

Initialise 500 individuals → pop.
while (<maximum fitness not found>)
    //choose a parent
    random member of pop → A.
    random member of pop → B.
    if (fitness(A)>fitness(B)) then
        A→parent1
    else
        B→parent1
    //create child
    mutate(parent1)→ child.
    //choose an individual to be replaced
    random member of pop → A.

```
        random member of pop → B.
        if (fitness(A)>fitness(B)) then
                child→B.
        else
                child→A.
```

This is a somewhat inelegant (but easy to follow) way to implement a steady state GA with tournament selection. It picks two individuals at random, mutates the fitter one. Then picks another two individuals and replaces the less fit one with the new child. You need to make this code to output the number of fitness evaluations used to find the perfect individual. You only need to count fitness evaluations that are not duplicate – an efficient implementation would store the fitness of an individual with each individual so that it didn't need to be calculated more than once – thus you only need to count one new evaluation per iteration through the while loop.

5) Crossover.

You need a routine that takes two individuals and produces a new one that has some of the characters from the first individual and some of the characters from the second individual. In uniform crossover, each separate gene (character) is picked with equal probability from either parent1 or parent2.

Informal pseudocode:

```
for i from 1 to L
        if (rand()<0.5) then
                parent1[i]→ child[i]
        else
                parent2[i] → child[i]
```

```
e.g.
Parent 0 .!5AhHaR?-Cfee0TarU@R{wga?Bc
Parent 1 @Rb|hg(q0!G<CG-D<!6#5QNsi%+T
Child    .!b|hgaq?-C<eG0D<rU#5QNga%+T
```

6) A GA with crossover

Finally, you're ready to do a GA with crossover.

```
Initialise 500 individuals → pop.
while (<maximum fitness not found>)
        //choose a parent
        random member of pop → A.
        random member of pop → B.
        if (fitness(A)>fitness(B)) then
                A→parent1
        else
                B→parent1
        //choose a second parent
        random member of pop → A.
        random member of pop → B.
        if (fitness(A)>fitness(B)) then
                A→parent2
        else
```

```
        B→parent2
//create child
crossover(parent1, parent2)→result_of_crossover.
mutate(result_of_crossover)→ child.
//choose an individual to be replaced
random member of pop → A.
random member of pop → B.
if (fitness(A)>fitness(B)) then
        child→B.
else
        child→A.
```

You can now compare the number of evaluations used for the GA without crossover against the GA with crossover.

Questions

So, is the hypothesis correct? Is it true that a genetic algorithm with crossover finds solutions to a simple optimisation problem faster when using crossover than without crossover?

- If you use a small population is the result the same? Explain.
- In this problem any simple optimisation method like these will always find the optimum solution sooner or later. If the problem was a more difficult optimisation problem, do you think the answer to the hypothesis would be the same? Explain.

Further questions for those that find the assignment so far to be easy:

- Does using a higher mutation rate affect the result? Why?
- Try a range of mutation rates for the mutation-only GA, and a range of mutation rates for the GA with crossover. Does the best mutation rate for the mutation-only GA work faster than the best mutation rate for the GA with crossover?
- How does the best of the best of the GAs compare to the speed of the mutation hill-climber?

[ii] Source code for this applet is also available at that site and several other places. Remember the point of this assignment is for you to understand what you need to understand in order to do later assignments. So, I hardly need to say that if you use code from this site without understanding it properly then you'll not be achieving this goal and you'll be starting from a disadvantaged position when you come to do those later assignments. You're literally only cheating yourself, as they say, because you're not even going to hand this in. In an assessed assignment, presenting code or anything else from a website or other source as your own work is, of course, plagiarism and formal disciplinary action will be taken against students that do so.